Developing VAX C Programs
on ULTRIX

Chapter 2

Developing VAX C Programs for ULTRIX

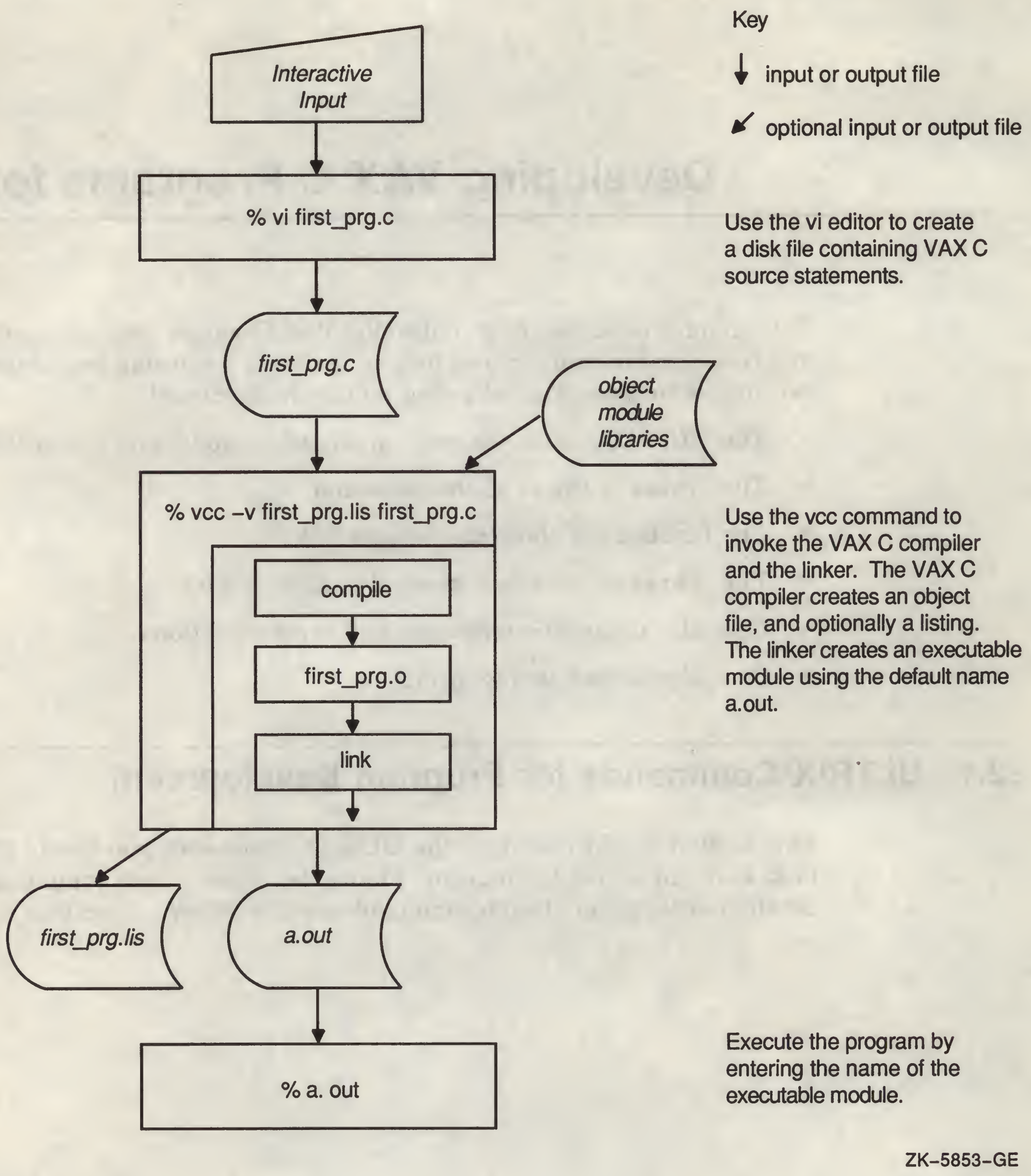
This chapter describes how to develop VAX C source programs and how to use the **vcc** command to compile and link your source programs into object modules and executable images. The following topics are discussed:

- The ULTRIX commands used to create, compile, and link a VAX C program
- The syntax of the vi editor command
- The functions of the compiler and linker
- The syntax of the **vcc** command and its options
- Compiler diagnostic messages and error conditions
- Compiler output listing format

2.1 ULTRIX Commands for Program Development

This section briefly describes the ULTRIX commands you use to create, compile, link, and run a VAX C program. Figure 2–1 shows these commands. For a more detailed description of each command, see the following sections.

Figure 2–1: Commands for VAX C Program Development



Key
input or output file
optional input or output file
Interactive Input
% vi first_prg.c
Use the vi editor to create a disk file containing VAX C source statements.
first_prg.c
object module libraries
% vcc –v first_prg.lis first_prg.c
compile
first_prg.o
link
Use the vcc command to invoke the VAX C compiler and the linker. The VAX C compiler creates an object file, and optionally a listing. The linker creates an executable module using the default name a.out.
first_prg.lis
a.out
% a. out
Execute the program by entering the name of the executable module.
ZK–5853–GE

The following example shows each of the commands from Figure 2–1 executed in sequence:

```
%vi first_prg.c
%vcc -v first_prg.lis first_prg.c
%a.out
```

To create a VAX C source program you must invoke a text editor. In the previous example, the vi editor was used to create a program entitled first_prg.c. The c file extension is generally used with C source programs.

After you create a VAX C source program using the vi editor, you can use the **vcc** command to generate an executable module. In most instances, both the VAX C compiler and the linker are invoked and controlled by the **vcc** command. The **vcc** command invokes the compiler, and if the compilation completes without fatal errors and you have not elected to bypass linking, it then invokes the linker. The object files created by the compiler and any linker options and object files specified on the **vcc** command line are passed to the linker.

The **vcc** command creates an optional listing file when you include the **–v** file-name option or **–Vlist=**file.lis option on the command line. Section 2.3.3 describes the **vcc** command and its options (including those linker options commonly specified on the **vcc** command).

The executable module, generated by the linker, has the default file name a.out. To execute this program, enter the name of the executable module at the ULTRIX prompt (%).

2.2 Creating a VAX C Program

The vi editor is a screen-oriented display editor that allows you to edit one window of text at a time. A window is a block of text about the size of your terminal screen. When you invoke the vi editor, it copies the file and stores it in an editing buffer. It then displays a window of text from the editing buffer on your screen. Use vi commands to alter the text in the editing buffer. When you finish changing the text, you end the editing session and the vi editor writes the editing buffer to a file.

To invoke the vi editor with the **vi** command, use the following syntax:

vi *filename*

For example, the following command edits, or creates, a file called myprogram.c:

```
%vi myprogram.c
```

After your file is open, you can use the vi commands to enter and edit text.

2.3 Compiling and Linking a VAX C Program

You can use the **vcc** command to compile and link your VAX C programs. This section discusses the **vcc** command and its options along with the functions performed by the VAX C compiler and the linker.

2.3.1 Functions of the Compiler

The primary functions of the VAX C compiler are as follows:

- Verify the correctness of C source statements and to issue informational, warning, and error messages
- Generate machine language instructions from the source statements of the C program
- Group these instructions into an object module that can be processed by the linker

The object file created by the compiler provides the linker with the following information:

- A list of all function, external, and global names declared in the program unit. The linker uses this information when it binds two or more program units together and must resolve references to the same names within the program units.
- A symbol table (if requested with the **–V debug** option or **–g** option on the **vcc** command line as described in Section 2.3.3.4). A symbol table lists the names of all external variables within a module, with definitions of their locations. The table is used in program debugging.

Section 2.3.2 describes the linker.

2.3.2 Functions of the Linker

Use the linker to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation. The linker's end product is an executable image that you can run on a VAX machine under the ULTRIX system environment.

Normally, you access the linker automatically when you enter the **vcc** command (unless you specify the **–V noobject** option or the **–c** option on the command line).

VAX C/ULTRIX has the capability to use two linkers: the ld linker and the lk linker. The ld linker is the default linker; the lk linker will only be invoked if **–V lkobject** is specified. See the *ULTRIX Reference Pages, Section 1* for more information on the ld linker. See Appendix A for more information on the lk linker.

2.3.3 The vcc Command

The **vcc** command program is your interface to the VAX C compiler. It accepts a list of file names and option switches and causes one or more processors (preprocessor, compiler, assembler, or linker) to process the files.

The **vcc** command has the following form:

vcc [–*options* [*args*]]... *filename*[.*type*] [...*filename*[.*type*]] [–*options* [*args*]]

***–options* [*args*]**
Indicates either special actions to be performed by the compiler or linker, or special properties of the input/output (I/O) files. See Section 2.3.3.4 for details about command-line options.

filename[.type]
Specifies the source files containing the program units to be compiled. If type is omitted or is not one of the following types, the file is assumed to be an object file and is passed directly to the linker. The following types have special meaning to the **vcc** command and are handled as follows:

.c or .h	Identifies files passed to the C compiler
.s	Identifies files passed to the ULTRIX assembler
.o or .a	Identifies files passed to the linker

You can specify more than one source file. If you specify more than one file, each compiles separately and the resulting object files are linked together to form one executable image.

2.3.3.1 Usage Considerations

In many instances it is enough to specify the name of the VAX C source file on the **vcc** command line without specifying any of the command line options. The VAX C compiler processes the file and passes the resultant object file to the linker along with the appropriate run-time libraries. The linker then produces an executable image with the default file name a.out.

You can select from a variety of processing options and specify files other than VAX C source files. The combination of the processing options and the type of each file determines how the **vcc** command handles the processing. If the options that you specify do not negate a certain level of processing, the **vcc** command examines the type of each file and passes the file to one or more of the following processors: the VAX C compiler, the ULTRIX assembler, or the linker.

You can compile a file for a specific system–SYSTEM_FIVE, BSD, or POSIX–by appending the **–Y** option to the **vcc** command or by setting the PROG_ENV environment variable. The **–Y** option defaults to SYSTEM_FIVE; the environment variable defaults to BSD. A **–Y** option specification overrides an environment variable specification. If neither **–Y** nor PROG_ENV is specified, the default is SYSTEM_FIVE. See Table 2–1 for more information on the **–Y** option. See the *ULTRIX Reference Pages, Section 3* for more information on the PROG_ENV environment variable.

If **–YSYSTEM_FIVE** is specified, the **–DSYSTEM_FIVE** parameter is added to the vaxc command and the **–YSYSTEM_FIVE** parameter is added to the linker call. In addition, the linker parameters **–lc**, **–lcg**, or **–lc_p** are preceded by **–lcV**, **–lcVg**, or **–lcV_p** and the linker parameters **–lm**, **–lmg**, or **–lmp** are changed to **–lmV**, **–lmVg**, or **–lmV_p**. Similarily, if **–YPOSIX** is specified, the **–DPOSIX** parameter is added to the vaxc command, the **–YPOSIZ** parameter is added to the linker call, and the linker parameters **–lc**, **–lcg**, or **–lc_p** are preceded by **–lcP**, **–lcPg**, or **–lcP_p**. Other parameters remain unchanged. If **–YBSD** is specified, the parameter **–YBSD** is added to the linker call.

You can also link routines written and compiled in other languages with VAX C source code. For example, suppose that you have a utility routine written in VAX FORTRAN that you want to call from a VAX C program. The VAX C program is contained in a file named myprog.c. The VAX FORTRAN routine is named utilityx and is contained in a file named utilityx.f. You write a JBL program defining a jacket for calling utilityx, and name it uxjacket.jbl. If you want to link the object files of both the FORTRAN routine and the JBL program to the C program, you can enter the following **vcc** command:

```
% vcc --V lkobject myprog.c utilityx.o uxjacket.o
```

This command processes the files as follows:

- Compiles myprog.c with the VAX C compiler
- Uses the lk linker to link the three routines together into the executable program using the default file name a.out

You can specify standard input as well. VAX C accepts input from standard input with the following syntax, which runs the compile directly:

```
% /usr/lib/vaxc sys%input
```

The output file is named sys$input.o.

If you want to insert compiled programs into a library instead of linking them directly into an executable program, use the **–c** option on the **vcc** command. See Section 2.3.3.4 for more information.

For example, to compile three C source files, a.c, b.c, and c.c, into .o files for insertion into a library, enter the following command:

```
% vcc -c a.c b.c c.c
```

This command compiles the programs and generates the output files a.o, b.o, and c.o.

The following diagram shows how the **vcc** command program processes the various types of files (unless a specified command line option negates some part of the processing):

```
x.s          -> [as]       -> x.o -> [linker] -> a.out

x.c, x.h     -> [vcc -c]   -> x.o -> [linker]-> a.out

x.o, x.a                          -> [linker]-> a.out
```

Files types that are not recognized are treated as object files and passed to the linker. If you specify more than one file type on the command line, the **vcc** command processes each file according to its type.

If the linker produces an a.out file, the **vcc** command program deletes the object files that it created. Object files specified on the **vcc** command line with the **–o** option are retained.

2.3.3.2 Specifying Input Files

Include the full input file specification on the **vcc** command line. If the files are not in your current working directory, you must specify their directory locations.

If you specify multiple files, you must separate the file specifications by spaces. Commas and other special characters are not interpreted as file separators; they are interpreted as part of the file name.

2.3.3.3 Specifying Output Files

The output produced by the compiler includes the following file types:

- An object file when the **–c** option is specified on the command line
- An executable file unless the **–c** option is specified on the command line
- A listing file when the **–v** *filename* option or the **–V list=***filename* option is specified on the command line

You can control the production of these files by specifying the appropriate options on the **vcc** command line. If the files are successfully linked, the **vcc** command program deletes the .o files created by the compiler.

NOTE

Section 2.3.3.4 describes the command line options in detail.

For VAX C files, the compiler generates an object file (.o file type) by default.

During the early stages of program development, you may find it helpful to use the **–c** option to suppress the production of executable modules until your source program compiles without errors. If you do not specify the **–c** option, the compiler generates executable modules as follows:

- If you specify one source file, one object file is generated and is passed to the linker.
- If you specify multiple source files, separated by spaces, each source file is compiled separately and an object file is generated for each source file. The linker is then invoked to link all files into a single executable file.

To produce an executable file with an explicit file specification, you must use the **–o** option (see Section 2.3.3.4). Otherwise, the object file has the name a.out.

The following examples show two **vcc** commands. Each command is followed by a description of the output files that it produces.

```
% vcc -V list=aaa.lis aaa.c bbb.c ccc.c
```

The VAX C source files (aaa.c, bbb.c, and ccc.c) are compiled as separate files. The object files are then passed to the linker, producing the executable file a.out and the listing file aaa.lis. The **–V** option specifies the listing file name on the command line.

```
% vcc -c circle.c
```

The VAX C source file circle.c is compiled, producing an object file named circle.o. The object file is not passed to the linker because the **–c** option is specified.

2.3.3.4 Options to the vcc Command

The **vcc** command options influence how the compiler processes a file. In many cases, the simplest form of the **vcc** command is sufficient. The options are necessary only when special processing is required.

NOTE

If you include the **–g** option on the **vcc** command line to prepare a file for use with the dbx debugger, optimization is turned off. Optimizations performed by the compiler can cause unexpected behavior when you are using the dbx symbolic debugger.

Table 2–1 summarizes the **vcc** command options.

Table 2–1: Options Supported by the vcc Command

Option	Description
–B *string*	Finds a substitute compiler,a preprocessor, an assembler, or a linker in the files named by *string*. If *string* is empty, use a standard backup version.
–b	Does not pass the library file –lc to the linker.
–c	Generates an object file with a .o file extension. The linked, executable module is not generated.
–D *name=def*	Assigns the specified value (*def*) to *name*. The preprocessor interprets this option. If a definition value is not specified, the name is set equal to 1.
–E	Runs only the vcc preprocessor and sends the result to standard output. The code is preprocessed and all preprocessor directives, such as include file statements, are resolved. The compiler and the linker are not invoked.
–Em	Runs the cpp preprocessor and produces the makefile dependencies. The file is preprocessed; it is not compiled or linked.
–f	Enables single-precision, floating-point arithmetic. Double-precision, floating point arithmetic is the default selection. Procedure arguments are still promoted to double-precision, floating point format.
–g	Generates additional symbol table information for the dbx debugger.
–I*dir*	Specifies the name of the directory containing the relevant include files. A search for included files whose names do not include a directory specification occurs in the directory of the file, the directory named by the **–I** option, and finally in the directories contained in a standard list.
–K	Generates a full MAP table. This is a linker option. It may be specified on the **vcc** command line or the linker command line.
–l*x*	Specifies a library to include in the link process. The variable *x* is an abbreviation for the library and path name /lib/libx.a in which *x* is a string. If the library is not found, the linker searches for /usr/local/lib /libx.a. A library search starts when the library name is encountered. As a result, the placement of the **–l** within the **vcc** or linker command line is significant.
–Md	Specifies the floating-point type as **D_FLOAT** double-precision, floating-point format. This is the default selection. In addition, the linker receives the **–lc** flag.
–Mg	Specifies the floating-point type as **G_FLOAT** double-precision, floating-point format. In addition, the linker receives the **–lcg** flag. If you want to use the math library, with code generated with the **–Mg** option, you must link in the G_FLOAT version of the library by specifying **–lmg** on the linker or **vcc** command line.
–o	Accepts the specified name as the final output file name. This is a linker option. It may be specified on the **vcc** command line or the linker command line.
–O	Invokes the object code improver. The default selection is to perform object code optimization. See Section 2.3.3.5 for information on how to turn off optimization.

(continued on next page)

Table 2–1 (Cont.): Options Supported by the vcc Command

Option	Description
–p,**–pg**	Prepares object files for profiling. The **–pg** option also invokes a run-time recording mechanism that produces a gmon.out file. This file contains more extensive statistics.
–t [0pal]	Finds only the designated compiler, preprocessor, assembler, or linker in the files whose names are constructed by a **–B** option. In the absence of a **–B** option, these are found in the standard places.
–U*name*	Makes the specified variable undefined within the program. This option is interpreted by the preprocessor.
–v *filename.lis*	Produces the listing file, complete with a cross-reference table and machine code listing.
–V *option*	Compiles the source code using vendor-specific options. Section 2.3.3.5 lists these options.
–w	Suppresses compiler warning messages. Error messages are displayed, but warning messages are not.
–Y[param]	Compiles a file for one of the following systems: SYSTEM_FIVE, BSD, or POSIX. If a parameter other than SYSTEM_FIVE, BSD, or POSIX is specified, a warning is printed and the **–Y** option is ignored. If no parameter is specified, **–Y** defaults to **–YSYSTEM_FIVE**. If multiple **–Y** options are specified, only the last option takes effect, and no warning message is generated. Section 2.3.3.1 describes how these parameters can be set with the PROG_ENV variable and describes linkage considerations.

2.3.3.5 Specifying Vendor-Specific Options

You can use the **–V** option on the **vcc** command line to specify a vendor-specific option. This option accepts DCL-type VAX C compilation control qualifiers as arguments. (DCL, the Digital Command Language, is offered by Digital on the VMS operating system.)

The syntax for the **–V** option is as follows:

–V *args*

The args are either a quoted string of arguments separated by spaces or a series of arguments separated by commas (with no spaces). The arguments are not case sensitive, and you can abbreviate and specify them in any order. You can use the qualifier names and abbreviations from a DCL CC command line as **–V** option arguments. However, do not use slashes (/) because they conflict with ULTRIX file names.

For example, you can enter the following DCL CC command line in the VMS environment:

```
CC /CROSS_REFERENCE/SHOW=INCLUDE/MACHINE_CODE/LIST FOO.C
```

The following two command lines are the VAX C equivalents to the preceding command line within the ULTRIX operating system environment:

```
% vcc -v foo.l -V show=include foo.c
% vcc -V Cross_reference,show=include,list=foo.lis foo.c
```

The command name, file names, and option switches remain case sensitive.

Table 2–2 lists the DCL type options that you can use as arguments to the **vcc** command's **–V** option. These option arguments are described in detail in the list following the table.

Table 2–2: Arguments to the –V Option of the vcc Command

Argument	Argument Values	Negative Form	Default
cross_reference	—	**nocross_reference**	**nocross_reference**
debug=	**[no]traceback**	**nodebug**	**nosymbols**
	[no]symbols		
	all		**traceback**
	none		
define	*definition-list*	**nodefine**	**nodefine**
g_float	—	**nog_float**	**nog_float**
list[=*file-spec*]	*file-spec*	**nolist**	**nolist**
lkobject	none	**nolkobject**	**nolkobject**
machine_code=	**interspersed**	**nomachine_code**	**nomachine_code**
object[=*file-spec*]	*file-spec*	**noobject**	**object**
optimize=	**[no]inline**	**nooptimize**	**optimize**
show=	**[no]intermediate**	**noshow**	**noshow**
	[no]brief		
	[no]expansion		
	[no]include		
	[no]symbols		
	source		
	terminal		
standard=	**portable**	**nostandard**	**nostandard**
undefine	*undefine-list*	**noundefine**	**noundefine**
warnings=	**nowarnings**	**warnings**	
	noinformationals		

[no]cross_reference

The **cross_reference** argument to the **–V** option specifies that the storage map section of the listing file includes information about the use of symbolic names. The cross-reference contains the numbers of the lines in which the symbols are defined and referenced.

The option argument has the following form:

```
–V [no]cross_reference
```

The **cross_reference** argument is ignored if you do not generate a listing file, and you do not specify **–V show=symbols**.

The default is **nocross_reference**.

See Section 2.5.3 for a description of the listing format that is generated when you specify the **cross_reference** argument.

[no]debug

The **debug** argument to the **–V** option causes the compiler to provide information for use by the dbx symbolic debugger.

The option arguments have the following form:

```
–V debug=all
–V "debug=([no]symbols,[no]traceback)"
–V debug=none
–V nodebug
```

all
Directs the compiler to provide both local symbol definitions and an address correlation table.

symbols
Directs the compiler to provide the debugger with local symbol definitions for user-defined variables, arrays (including dimension information), structures, and labels of executable statements.

traceback
Directs the compiler to provide an address correlation table so that the debugger can translate virtual addresses into source program routine names and compiler-generated line numbers.

none
Directs the compiler to provide no debugging information.

If you do not specify the **debug** argument with the **–V** option on the **vcc** command line, the default is **debug=traceback**. Note that **debug** is the equivalent of **debug=all**, and **nodebug** is the equivalent of **debug=none**.

See Chapter 3 and the *Guide to Languages and Programming* for more information on debugging and traceback.

[no]define
The **define** argument to the **–V** option allows you to equate an identifier with a token string or a macro from the command line. Command line definitions override any internal definition statement. Similarly, **undefine** revokes a previous definition.

NOTE

If the **define** and **undefine** options are both present on the command line, the **define** statement is resolved first.

The option argument has the following form:

–V define=*(identifier[=definition][,...])*

identifier
Specifies the identifier to be defined.

definition
Specifies the value to be associated with the identifier.

The **define** and **undefine** options are functionally equivalent to the **#define** and **#undef** preprocessor directives. The simplest form of the **define** option equates an identifier with the default value, which is 1. For example:

```
-V define=true
```

This command option is equivalent to the following source code statement:

```
#define  true  1
```

You must enclose macro definitions in quotation marks ("). For example:

```
-V define="funct(a)=a+sin(a)"
```

This command option is equivalent to the following source code statement:

```
#define  funct(a)  a+sin(a)
```

If you use quotation marks within your definition, a space or a single equal sign is interpreted as a delimiter. Consider the following example:

```
-V define="new_val=51"
```

This command option is equivalent to the following source code statement:

```
#define  new_val  51
```

However, the following command line and source code definitions are equivalent:

```
-V define="new_val =51"
#define  new_val  =51
```

If you do not use quotation marks within your definition, an equal sign is the only recognized delimiter. A space indicates the end of the definition. Consider the following example:

```
-V define=(test1=4,"funct(x)=(a+b)/x")
```

This command option is equivalent to the following source code statements:

```
#define  test1  4
#define  funct(x)  (a+b)/x
```

In the following command line, the definition is not accepted due to the improper use of spaces. In such an instance, an erroneous attempt is made to interpret the information as a file specification instead of part of the definition. This occurs because the space in the statement ends the definition portion of the command line.

```
define= new_val=10
```

As the previous examples show, if you use quotation marks, the compiler interprets either the first space or the first equal sign as a delimiter character. If you do not use quotation marks, the compiler treats the first equal sign as a delimiter. In both instances, any additional equal signs are considered to be part of the value. Thus, special care is required when specifying an equal sign in a value. The following examples show three possible ways to specify an equal sign so that it is treated as part of the value rather than as a delimiter:

```
-V define=(equ==)
-V define=("equa =")
-V define=("equal==")
```

In the first definition, which does not use quotation marks, the first equal sign is a delimiter and the second one is part of the defined value. In the second definition, the space is the delimiter, so the single equal sign is accepted as part of the value. The third example is similar to the first, except that it shows the use of the double equal sign within quotation marks.

[no]g_float

The **g_float** argument to the **–V** option controls how the compiler implements objects of type **double**.

The option argument has the following form:

–V [no]g_float

The default, **nog_float**, causes the compiler to implement double-precision quantities using the VAX D_floating-point data type. Specifying **g_float** causes the compiler to implement double-precision values using the VAX G_floating-point data type.

If your program requires the G_floating-point type form of double-precision data for its correct operation (that is, it uses a range larger than 10^{38}), specify the **g_float** option. See Chapter 7 for more information about using **double** in VAX C.

Routines that pass and receive double-precision quantities should use the same data type as the routines that they pass data to or receive data from. For example, do not pass D_floating point data to a program that uses the G_floating point data type.

CAUTION

> VAX ULTRIX systems support both D_floating and G_floating implementations of type **double**. On different systems, however, the performance of a program can vary widely depending on whether your program is compiled with G_floating or D_floating. The difference exists when a particular system supports one floating-point type in hardware and the other in software. If you want to optimize performance, and if range and accuracy constraints do not dictate one of the two options, you must ensure that the most efficient option is in effect during the compilation.

See Chapter 7 for more information on floating-point data types.

[no]list

The **list** argument to the **–V** option specifies that a source listing file is to be produced.

The option argument has the following form:

–V list[=*file-spec*]

You can include a file specification for the listing file. If you do not, the listing file name defaults to the name of the first source file that you specify on your **vcc** command line. This file has the default file extension .lis. The listing file is not automatically printed. You must use the **lpr** command to obtain a line printer copy of the listing file.

Section 2.5.1 discusses the format of a source code listing.

[no]lkobject

The **lkobject** argument to the **–V** specifies that VMS object code format, rather than the default BSD object code format, should be generated for object files. VMS object files should be passed to the lk linker, rather than the default ld linker.

The option argument has the following form:

–V [no]lkobject

The **vcc** shell, by default, generates BSD .o format for its object files and passes these objects to the ld linker to be linked. If files with a .obj extension appear on the command line, or if **–V lkobject** argument is specified, the files are passed to the lk linker instead.

Object files produced by both **–V lkobject** and **–V nolkobject** have the .o file extension. However, the ld linker will not link files produced with **–V lkobject** or produced by versions of **vcc** prior to Version 4.0. The linker returns an error message stating that the symbol ILLEGAL_LD_OBJECT, USE_LK is undefined.

The default is **nolkobject**.

[no]machine_code
The **machine_code** argument to the **–V** option specifies that the listing file includes a symbolic representation of the object code generated by the compiler. Generated code is represented in a form similar to an assembly listing.

NOTE

> Do not try to assemble and run the object code in the listing file. The code shown in the listing file is similar to VAX MACRO source code; however, the listing file contains constructs not supported by the VAX ULTRIX assembler and is not intended to be used for this purpose.

The option argument has the following form:

```
–V [no]machine_code[=interspersed]
```

If **interspersed** is specified, the listing will consist of lines of source code followed by the corresponding lines of machine code.

If you do not generate a listing file, this option is ignored. The default is **noma-chine_code**.

Section 2.5.2 describes the format of a machine code listing.

[no]object
The **object** argument to the **–V** option specifies the name of the object file.

The option argument has the following form:

–V noobject
–V object=*file-spec*

The default is **object**, which generates an object file for linking. If you omit the file specification, the object file defaults to the name of the first source file with a .o file extension.

You can use the negative form (**noobject**) to suppress object code generation. This allows you to test for compilation errors in the source program.

[no]optimize[=[no]inline]
The **optimize** argument to the **–V** option specifies that the compiler is to produce optimized code. This optimization takes place within the compiler, not as a separate program. VAX C does not use the object code improver.

The option argument has the following form:

```
–V [no]optimize
```

When you specify **–V optimize=inline**, the compiler performs function inline expansion optimization.

The default is **optimize**. If you include the **–g** or **–V debug** option on your **vcc** command line, optimization is not performed. If you want to debug an optimized program, you must request optimizations with the **–O** or **–V optimize** options to the **vcc** command line.

[no]show
The **show** argument to the **–V** option controls whether or not listing options are selected.

The option argument has the following form:

```
–V show
–V noshow
–V show=all
–V show=[no]brief
–V show=[no]expansion
–V show=[no]include
–V show=[no]intermediate
–V show=none
–V show=[no]source
–V show=[no]symbols
–V show=[no]terminal
```

Use the **list** option with the **show** option to select or cancel any of the options in the previous list. For example, the following command line creates a listing file with the file name mylist.lis. This file includes a listing of the files referenced by the **#include** directive.

```
%vcc programname.c -V list=mylist.lis,show=include
```

all
Specifies that all information is to be included in the listing file.

[no]brief
Generates a listing similar to the one created with the **symbols** option. The only difference between the two is that the **brief** option eliminates any symbols that are not identified in the program, or that do not belong to any union or structure referenced by the program.

The default is **nobrief**.

[no]expansion
Includes the final macro expansions in the listing. When you use this option, the number of substitutions performed on the line is printed next to each line.

The default is **noexpansion**.

[no]include
Includes the modules referenced by the **#include** directives in the listing.

The default is **noinclude**.

[no]intermediate
Includes all intermediate and all final macro expansions in the listing.

The default is **nointermediate**.

none
Generates an empty listing file consisting of header information.

[no]source
Includes source statements in the program listing.

The default is **source**.

[no]statistics
Includes compiler performance statistics in the listing.

The default is **nostatistics**.

[no]symbols
Includes the symbol table in the program listing. The symbol table includes a list of all functions, the size and attributes of each variable, and a program section summary and function definition map.

The default is **nosymbol**.

[no]terminal
Displays compiler messages on the terminal.

The default is **terminal**.

Specifying the **show** argument without any parameters is equal to specifying **show=all**; specifying the **noshow** argument is equal to specifying **show=none**.

standard
standard=noportable
The **standard** argument to the **–V** option specifies that the compiler is to generate informational diagnostics about VAX C language extensions and C code constructs that represent a relaxation of standard C conventions and rules.

The option argument has the following form:

```
–V standard
–V standard=portable
–V standard=noportable
```

You can specify the **standard** option with or without the variable portable; the results are the same. The **standard** option causes the compiler to generate warning messages when it encounters coding practices that are contrary to standard C. This increases portability between VAX C and other implementations of the C language. (This qualifier will not provide warnings about ANSI C features not supported by pcc.)

The default is **standard=noportable**.

If you specify the **nowarnings** argument to the **–V** option, the **standard** argument is ignored.

[no]undefine
The **undefine** argument to the **–V** option allows you to revoke a previous definition. If the **define** and **undefine** options are both present on the command line, the **define** statement is resolved first.

The option argument has the following form:

–V define=*identifier[,identifier ...]*

identifier
Specifies the identifier chosen to have its definition revoked. You can specify a list of identifiers separated by commas.

The **define** and **undefine** options are functionally equivalent to the **#define** and **#undef** preprocessor directives. The **define** option is described earlier in this section.

[no]warnings
The **warnings** argument to the **–V** option causes the compiler to generate informational (I) and warning (W) diagnostic messages in response to informational and warning-level errors.

The option argument has the following form:

```
–V warnings
–V warnings=noinformational
–V warnings=nowarnings
–V nowarnings
```

warnings
The compiler generates informational and warning diagnostic messages. An informational message indicates that a correct C statement may have unexpected results or may contain nonstandard syntax or source forms. A warning message indicates that the compiler detected acceptable, but nonstandard, syntax or performed some corrective action; in either case, unexpected results may occur. To suppress I and W diagnostic messages, specify the negative form of this argument (**nowarnings**).

The default is **warnings**.

warnings=noinformational
The compiler suppresses informational messages. Warning messages are still displayed. The default prints these messages.

warnings=nowarnings
The compiler suppresses all warning messages. Informational messages are still displayed. The default prints these warnings.

nowarnings
The compiler suppresses all messages except for the informational message SUMMARY. The default prints all messages.

Appendix B discusses compiler diagnostic messages.

2.4 Compiler and Linker Diagnostic Messages

Both the compiler and the linker issue error messages that help you to isolate the cause of an error condition. The following sections discuss these messages and error conditions.

2.4.1 Compiler Diagnostic Messages and Error Conditions

One of the functions of the C compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to the stderr output file and to any listing file. If you enter the **vcc** command interactively, the messages are displayed on your terminal. A message from the compiler has the following format:

```
"filename." line nnn: %severity-mnemonic, msg
```

The VAX C compiler replaces the severity code that precedes the message text with one of the following characters:

F	Signifies a fatal condition message
I	Signifies an informational message
W	Signifies a warning message
E	Signifies an error message

S Signifies a success message

Diagnostic messages usually provide enough information for you to determine the cause of an error and correct it.

Each compilation with messages terminates with a summary indicating the number of error, warning, and informational messages generated by the compiler. The summary has the following form:

```
"filename" line nnn: completed with n error(s), n warning(s), and n informational messages.
```

If the compiler creates a listing file, it also writes the messages to the listing file. Messages typically follow the statement causing the error.

Appendix B contains additional information about diagnostic messages, including descriptions of the individual messages.

2.4.2 Linker Diagnostic Messages and Error Conditions

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any errors or fatal conditions occur (severities E or F), the linker does not produce an image file.

Linker messages are descriptive, and you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking are as follows:

- An object module has compilation errors. This error occurs when you attempt to link a module that had warnings or errors during compilation. Although you can usually link compiled modules for which the compiler generated messages, verify that the modules will produce the output that you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one main program has been defined. The image file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names from the **lk** or **ld** command and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link modules, you can often correct it by reentering the command string and specifying the correct routines or libraries.

2.5 Compiler Listings

An output listing produced by the VAX C compiler consists of the following sections:

- A source code section
- A machine code section (optional)
- A storage map section (cross-reference, optional)

Sections 2.5.1 through 2.5.3 describe the compiler listing sections in detail.

2.5.1 Source Code Section

The source code section of a compiler output listing displays the source program as it appears in the input file, with the addition of sequential line numbers generated by the compiler. Example 2–1 shows a sample of a source code section from a compiler output listing.

Example 2–1: Sample Listing of Source Code

```
.MAIN
V1.0

  1           #include "stdio.h"
113           #define STARTNUM 1
114           #define ENDNUM 200
115
116
117           main ()
118           {
119     1         int    Count;
120     1         long   Square;
121     1
122     1         printf ("Table of squares \n\n");
123     1
124     1         for (Count = STARTNUM; Count <= ENDNUM ; Count++){
125     2             Square = (long)Count * (long)Count;
126     2
127     2             printf ("Number: %d Square: %ld\n", Count, Square);
128     2             }
129     1         }
130

Command Line
------------

/usr/lib/vaxc test.c -V list=test.lis
```

Compiler-generated line numbers appear in the left margin.

Compile-time and run-time error messages that contain line numbers refer to these compiler-generated line numbers. (See Appendix B for a summary of error messages.)

2.5.2 Machine Code Section

The machine code section of a compiler output listing provides a symbolic representation of the compiler-generated object code. The representation of the generated code and data is similar to that of an assembly listing. As an option, you can choose to intersperse the machine code with the source code for easier reading.

NOTE

The machine code is represented in VAX MACRO source code. This code is only for your reference so do not assemble and run it.

The machine code section is optional. To receive a listing file with a machine code section, you can choose one of the following three options:

–v *file.lis programname*
–V list=*file.lis*,machine_code *programname*
–V list=*file.lis*,machine_code=interspersed

Example 2–2 shows a sample of a machine code section from a compiler output listing.

Example 2–2: Sample Listing of Machine Code

```
                        0000 main:
                   0040 0000        .entry main, ^m<r6>
             5E 04 C2   0002        subl2  #4,sp
   56 00000000 EF 9E    0005        movab  $CHAR_STRING_CONSTANTS,r6
                66 DF   000C        pushal (r6)
  00000000* EF 01 FB    000E        calls  #1,printf
             5C 01 D0   0015        movl   #1,ap
                        0018 sym.1:
          52 5C 5C C5   0018        mull3  ap,ap,r2
                52 DD   001C        pushl  r2
             14 A6 DF   0020        pushal 20(r6)
  00000000* EF 03 FB    0023        calls  #3,printf
E6 5C 000000C8 8F F3    002A        aobleq #200,ap,sym.1
             50 01 D0   0032        movl   #1,r0
                   04   0035        ret
                   04   0036        ret
```

The machine code follows the source code listing unless machine_code=interspersed is specified. The object module location of each statement and the machine code instructions are listed. The assembly language code, generated by each line of source text, is shown next to the corresponding machine code instruction.

The following summary outlines the conventions used to represent code and data in machine code listings:

- The VAX MACRO mnemonics represent the generated code.
- R0 through R11 represent the VAX general-purpose registers (0 through 11). Register 12 is an argument pointer that is represented by AP. Register 13 serves as the frame pointer, represented by the mnemonic FP. Register 14 is the stack pointer, represented by SP. Register 15 is the program counter, represented by PC.
- The compiler may generate labels for its own use.
- Signed integer values represent numeric constants.
- The function name plus the hexadecimal offset within that function represent the addresses. Changes from one function to another are indicated by .entry lines.

2.5.3 Storage Map Section

The storage map section of the compiler output listing is printed after each program unit, or function. It summarizes information in the following categories:

- External Declarations Section. The storage map lists all the names declared or defined outside of any function.

- Functions. The storage map lists all the functions in the source program. Along with each name, the following information is listed:
  - The identifier of the name
  - The line on which the name is declared
  - The size of the identifier
  - The storage class of the name
  - The data type of the name
- Function Definition Map. This portion of the storage map lists each function defined in the program and the line number that the function is defined on.

A heading for an information category appears on the listing only when entries are generated for that category.

Cross-reference information is optional. To obtain it, enter the **vcc** command with one of the following options:

–v *filename.lis programname*
–V list,cross_reference *programname*

When you request cross-referencing, the compiler lists the line number where each name is referenced.

Example 2–3 shows a sample storage map section with cross-reference information.

Example 2–3: Sample Storage Map Section

```
.MAIN.                  13-APR-1988 10:35:55    VAX C      V1.0-001     Page 3
V1.0                                                                 test.c (1)

                                   +-------------+
                                   | Storage Map |
                                   +-------------+

External Declarations
---------------------

 Identifier Name    Line    Size            Class        Type and References
 ---------------    ----    ----            -----        -------------------
```

(continued on next page)

```
ctermid             99                       Extern        Function returning
                                                           pointer to char
                                                              - No references
cuserid             99                       Extern        Function returning
                                                           pointer to char
                                                              - No references
freo                96                       Extern        Function returning
                                                           pointer to struct _iobuf
                                                              - No references
ftell               97                       Extern        Function returning
                                                           long int
                                                              - No references
gets                99                       Extern        Function returning
                                                           pointer to char
                                                              - No references
main                117                      Extern def.   Function returning
                                                           long int
                                                              - No references
popen               96                       Extern        Function returning
                                                           pointer to struct _iobuf
                                                              - No references
rewind              98                       Extern        Void function
                                                              - No references
setbuf              98                       Extern        Void function
                                                              - No references
setbuffer           98                       Extern        Void function
                                                              - No references
setlinebuf          98                       Extern        Void function
                                                              - No references
sprintf             102                      Extern        Function returning
                                                           pointer to char
                                                              - No references
temam               100                      Extern        Function returning
                                                           pointer to char
                                                              - No references
tmpnam              100                      Extern        Function returning
                                                           pointer to char
                                                              - No references
_iob                67      60 bytes         Extern        Array [3] of struct
                                                           _iobuf
                                                              - No references
_iobuf              60      20 bytes                       Structure tag
                                                              - Referenced at line 96
  _cnt              61      1 longword                     Member (offset = 0),
                                                           long int
                                                              - No references
  _ptr              62      1 longword                     Member (offset = 4 bytes),
                                                           pointer to char
                                                              - No references
  _base             63      1 longword                     Member (offset = 8 bytes),
                                                           pointer to char
MAIN.                 13-APR-1988 10:35:55     VAX C       V1.0-001        Page 4
V1.0                                                                    test.c (1)

  Identifier Name   Line    Size             Class         Type and References
  ---------------   ----    ----             -----         -------------------
```

(continued on next page)

Example 2–3 (Cont.): Sample Storage Map Section

```
 _bufsiz            64      1 longword                     Member (offset = 12 bytes),
                                                           long int
                                                              - No references
 _flag              65      1 word                         Member (offset = 16 bytes),
                                                           short int
                                                              - No references
 _file              66      1 byte                         Member (offset = 18 bytes),
                                                           char
                                                              - No references

Function "main" defined at line 117
-----------------------------------

 Identifier Name    Line    Size             Class         Type and References
 ---------------    ----    ----             -----         -------------------

Count               119     1 longword       Register      Long int
                                                              - Referenced at
                                                              lines 124(3),
                                                              125(2), and 127
printf              122                      Extern        Function returning
                                                           long int
                                                              - Referenced at
                                                              lines 122 and 127
Square              120     1 longword       Not Alloc.    Long int
                                                              - Referenced at
                                                              lines 125 and 127

Function Definition Map
-----------------------

 Line   Name
 ----   ----

117     main
Command Line
------------
/usr/lib/vaxc -v test.l -V "cross show=symbol machine" test.c
```

2.6 The lk Linker Image Map

The ld linker cannot generate an image map listing, but the lk linker can. An lk Linker Image Map consists of the following parts:

- An object module synopsis
- A program section synopsis
- A symbol cross-reference
- A symbol value listing
- An image synopsis
- A link-run statistics synopsis

An image map is generated when you specify the **–K** and **–V lkobject** options on the **vcc** command line or the **–K** option on the **lk** command line.

2.6.1 Object Module Synopsis

The Object Module Synopsis shows which object modules (files or library elements) were linked into the program image. Example 2–4 is a sample Object Module Synopsis from the lk Linker Image Map.

Example 2–4: Object Module Synopsis

```
 ❶                                  ❷                       ❸
 a.out             13-APR-1988 13:37     VAX ULTRIX Linker V2.0       Page   1

                                   +------------------------+
                                   ! Object Module Synopsis !
                                   +------------------------+

   ❹         ❺      ❻       ❼        ❽               ❾
Module Name  Ident    Bytes   File          Creation Date        Creator
-----------  -----    -----   -----         --------------       -------
crt0                    124  /lib/crt0.o   24-OCT-1986 16:47  Unknown ULTRIX Compiler
.MAIN.       V1.0       157  test.obj      25-NOV-1986 10:51  VAX C
printf                   80  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
doprnt                 2192  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
flsbuf                  296  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
data                    216  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
fclose                  260  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
close                    16  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
getstdiobuf             132  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
fstat                    16  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
isatty                   32  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
gtty                     24  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
exit                     20  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
_exit                     4  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
cerror                   12  /lib/libc.a   24-OCT-1986 10:51  Unknown ULTRIX Compiler
$$COMSY,S                 8  /lib/libc.a   24-OCT-1986 10:51  VAX ULTRIX Linker
```

Key to Example 2–4:

❶ File name of the program image.

❷ Date and time that the lk linker was run.

❸ Version number of the lk linker.

❹ Name of the object modules.

❺ Module ID, if one is specified.

❻ Size of the module, in bytes (decimal).

❼ File the module was read from.

❽ Date the file was created.

❾ Language processor that created the file, or "unknown ULTRIX compiler" if not known.

2.6.2 Program Section Synopsis

The Program Section Synopsis shows the layout of program sections (psects) and the modules that contributed to them in virtual memory. Example 2–5 is a sample Program Section Synopsis from the lk Linker Image Map.

Example 2–5: Program Section Synopsis

```
a.out            13-APR-1988 13:37       VAX ULTRIX Linker V2.0      Page   2
                                  +--------------------------+
                                  ! Program Section Synopsis !
                                  +--------------------------+

    ❶          ❷            ❸       ❹       ❺                   ❻       ❼
Psect Name  Module Name   Base     End     Length              Align   Attribute
----------  -----------   ----     ---     ------              -----   ----------

ULT$TEXT                00000000 00000F13 00000F14 (  3860.)  LONG 2  PIC,USR,CON
            crt0        00000000 0000004B 0000004C (    76.)  LONG 2
            printf      0000004C 00000073 00000028 (    40.)  LONG 2
            doprnt      00000074 00000903 00000890 (  2192.)  LONG 2
            flsbuf      00000904 00000A03 00000100 (   256.)  LONG 2
            data        00000A04 00000A77 00000074 (   116.)  LONG 2
            fclose      00000A78 00000B53 000000DC (   220.)  LONG 2
            close       00000B54 00000B63 00000010 (    16.)  LONG 2
            getstdiobuf 00000B64 00000BBB 00000058 (    88.)  LONG 2
            fstat       00000BBC 00000BCB 00000010 (    16.)  LONG 2
            isatty      00000BCC 00000BEB 00000020 (    32.)  LONG 2
            gtty        00000BEC 00000C03 00000018 (    24.)  LONG 2
            ioctl       00000C04 00000C13 00000010 (    16.)  LONG 2
            malloc      00000C14 00000E4B 00000238 (   568.)  LONG 2
            bcopy       00000E4C 00000EAF 00000064 (   100.)  LONG 2
            sbrk        00000EB0 00000EDF 00000030 (    48.)  LONG 2
            write       00000EE0 00000EEF 00000010 (    16.)  LONG 2
            exit        00000EF0 00000F03 00000014 (    20.)  LONG 2
            _exit       00000F04 00000F07 00000004 (     4.)  LONG 2
            cerror      00000F08 00000F13 0000000C (    12.)  LONG 2

$CODE$                  00000F14 00000F4A 00000037 (    55.)  LONG 2  PIC,USR,CON,REL
            .MAIN.      00000F14 00000F4A 00000037 (    55.)  LONG 2

$CHAR_STRING_CONSTANTS  00001000 0000102C 0000002D (    45.)  LONG 2  PIC,USR,CON,REL
            .MAIN.      00001000 0000102C 0000002D (    45.)  LONG 2

$DATA                   00001030 00001030 00000000 (     0.)  LONG 2  PIC,USR,CON,REL
            .MAIN.      00001030 00001030 00000000 (     0.)  LONG 2

ULT$DATA                00001030 0000119B 0000016C (   364.)  LONG 2  PIC,USR,CON,REL
            crt0        00001030 0000105F 00000030 (    48.)  LONG 2
            printf      00001060 00001087 00000028 (    40.)  LONG 2
            flsbuf      00001088 000010AF 00000028 (    40.)  LONG 2
            data        000010B0 00001113 00000064 (   100.)  LONG 2
            fclose      00001114 0000113B 00000028 (    40.)  LONG 2
            getstdiobuf 0000113C 00001167 0000002C (    44.)  LONG 2
            malloc      00001168 00001193 0000002C (    44.)  LONG 2
            sbrk        00001194 0000119B 00000008 (     8.)  LONG 2
_iob                    000010B4 000010EC 00000039 (    57.)  LONG 2  PIC,USR,CON,REL
            .MAIN.      000010B4 000010EC 00000039 (    57.)  LONG 2

ULT$COMM                0000119C 00001213 00000078 (   120.)  LONG 2  PIC,USR,CON,REL
            malloc      0000119C 0000120B 00000070 (   112.)  LONG 2
            $$COMSYMS   0000120C 00001213 00000008 (     8.)  LONG 2
```

Key to Example 2–5:

❶ Name of the program section.

❷ Names of the modules contributing to the program section.

❸ Starting virtual address (in hexadecimal) of the psect or module.

❹ Ending virtual address (in hexadecimal) of the psect or module.

❺ Length of the psect or module, given in both hexadecimal and decimal.

❻ Alignment specified for the psect. The numeric value is an integer from 0 to 9 that specifies the alignment as a power of two, as shown in the following table:

Value	**Alignment**
0	1 (BYTE)
1	2 (WORD)
2	4 (LONGWORD)
3	8 (QUADWORD)
4	2^4
...	...
9	2^9 (512 bytes)

Alignment on 512-byte boundaries, which is a half page for ULTRIX systems, is the maximum for a psect.

❼ Program section attributes (see Appendix A for more information).

2.6.3 Symbol Cross-Reference

The Symbol Cross-Reference lists symbolic names alphabetically, giving the value of the symbol, the module that defines it, and the modules that refer to it. Example 2–6 is a sample Symbol-Cross Reference table from the lk Linker Image Map.

Example 2–6: Symbol Cross Reference

```
a.out          31-JAN-1987 13:37        VAX ULTRIX Linker V2.0         Page   3
                                   +------------------------+
                                   ! Symbol Cross-Reference !
                                   +------------------------+
❶                    ❷          ❸                ❹
 Symbol               Value         Defined By        Referenced By ...
 ------               -----         ----------        -----------------
__cleanup            00000A68-R     data             exit
__doprnt             00000174-R     doprnt           printf
__exit               00000F04-R     _exit            exit
__flsbuf             00000904-R     flsbuf           doprnt
__fwalk              00000A04-R     data
__getstdiobuf        00000B64-R     getstdiobuf      flsbuf
__iob                000010B4-R     data             flsbuf        printf
__iobend             0000120C-R     $$COMSYMS        data
__iobstart           000010F0-R     data
_bcopy               00000E4C-R     bcopy            malloc
_close               000005BC-R     close            fclose
_edata               0000119C       <Linker>
 _end                00001214       <Linker>
_environ             00001034-R     crt0
_errno               00001210-R     $$COMSYMS        cerror
_etxt                00000F4B       <Linker>
_exit                00000EF0-R     exit             crt0
_fclose              00000AEC-R     fclose           data
_fflush              00000A78-R     fclose
_free                00000D04-R     malloc           fclose
_fstat               00000BC4-R     fstat            getstdiobuf
_gtty                00000BEC-R     gtty             isatty
_ioctl               00000C0C-R     ioctl            gtty
_isatty              00000BCC-R     isatty           flsbuf
_main                00000F14-R     .MAIN.           crt0
_malloc              00000C14-R     malloc           getstdiobuf
_moncontrol          00000044-R     crt0
_printf              0000004C-R     printf           .MAIN.
_realloc             00000D38-R     malloc
_realloc_srchlen     0000116C-R     malloc
_sbrk                00000EB0-R     sbrk             malloc        flsbuf
_write               00000EE8-R     write            fclose        close
cerror               00000F08-R     cerror           _exit         sbrk

curbrk               00001198-R     sbrk
mcount               00001038-R     crt0
minbrk               00001194-R     sbrk
start                00000000-R     crt0
```

Key to Example 2–6:

❶ Symbol name.

❷ Symbol value, in hexadecimal. See the listing of symbol values for the meaning of the letters suffixed to the values.

❸ Module that defined the symbols. Symbols defined internally by the lk linker display the value <Linker>.

❹ Modules that refer to the symbol (if any).

2.6.4 Symbol Value Listing

The Symbol Value Listing lists the values of the symbols and the symbolic names that have that value. Example 2–7 is a sample Symbol Value Listing from the lk Linker Image Map.

Example 2–7: Symbol Value Listing

```
a.out          13-APR-1988 13:37      VAX ULTRIX Linker V2.0        Page   4

                                  +-------------------+
                                  ! Symbols By Value !
                                  +-------------------+

   ❶                                      ❷
  Value                                  Symbols...
  -----                                  ----------
0000004C                                 R-_printf
00000174                                 R-__doprnt
00000904                                 R-__flsbuf
00000A68                                 R-__cleanup
00000AEC                                 R-_close
00000B64                                 R-__gestdiobuf
00000BC4                                 R-_fstat
00000BCC                                 R-_isatty
00000BEC                                 R-_gtty
00000C0C                                 R-_ioctl
00000C14                                 R-_malloc
00000D04                                 R-_free
00000E4C                                 R-_bcopy
00000EB0                                 R-_sbrk
00000EE8                                 R-_write
00000EF0                                 R-_exit
00000F04                                 R-__exit
00000F08                                 R-_cerror
00000F14                                 R-_main
000010B4                                 R-_iob
0000120C                                 R-_iob_end
00001210                                 R-_errno
00001214                                   _end

                         ❸
         Key for special characters above:
                +-------------------+
                ! *  - Undefined    !
                ! R  - Relocatable  !
                ! WK - Weak         !
                +-------------------+
```

Key to Example 2–7:

❶ Hexadecimal value.

❷ Symbols that have the hexadecimal value.

❸ Description of the special characters is as follows:

*	Undefined. (The symbol was not defined anywhere in the link.)

R	Relocatable. The definition of the symbol was calculated as an offset from a program section, and thus could change depending on the base virtual address that the lk linker assigns to that psect. If no R appears, the value of the symbol is independent of psect address assignment.
WK	Weak. If the lk linker encounters a reference to a symbol that is not defined anywhere, it normally reports this as an error. However, if the symbolic reference is a weak reference, the linker assigns the symbol a value of 0 and does not report an error. If, however, the linker reports any unresolved strong references, it also reports all unresolved weak references. A weak definition is not included in the _SYMTAB directory of a run-time library. When the linker is searching a library to resolve references (strong or weak), it will not select a module for inclusion on the basis of a weak definition of a symbol. However, if the linker has selected that module for inclusion on the basis of a strong reference to another symbol, it will resolve all references to weak definitions that may be present in that module.

2.6.5 Image Synopsis

The Image Synopsis is a summary of the entire link. Example 2–8 is a sample Image Synopsis from the lk Linker Image Map.

Example 2–8: Image Synopsis

```
a.out                  13-APR-1988 13:37       VAX ULTRIX Linker V2.0    Page   5
                                   +----------------+
                                   ! Image Synopsis !
                                   +----------------+
Virtual memory allocated:                      ❶ 00000000 00001213 00001214
                                                    (4628. bytes, 5. pages)
Text section virtual address limits:           ❷ 00000000 00000FFF 00001000
                                                    (4096. bytes, 4. pages)
Data section virtual address limits:           ❸ 00001000 000013FF 00000400
                                                    (1024. bytes, 1. page)
BSS section virtual address limits:            ❹ 00001400 00001400 00000000
                                                    (0. bytes, 0. pages)
Number of files:                                      5.
Number of modules:                                   21.
Number of program sections:                           8.
Number of global symbols:                            54.
Number of cross references:                          65.
User transfer address:                         ❺ 00000F14
Image type:                                     Demand-loadable (ZMAGIC)
```

Key to Example 2–8:

❶ Total virtual memory allocated to the program.

❷ Limits of the program text (executable) section.

❸ Limits of the initialized data section.

❹ Limits of the uninitialized data (BSS) section.

NOTE

For items 1 through 4, the low and high virtual address of the section and its length is shown in hexadecimal, decimal, and pages.

❺ Address, in hexadecimal, to which control is transferred when the program is run.

2.6.6 Link Run Statistics Synopsis

The Link Run Statistics Synopsis contains statistics and performance indicators for the linker run. Example 2–9 is a sample Link Run Statistic Synopsis from the lk Linker Image Map.

Example 2–9: Link Run Statistics Synopsis

```
                            +----------------------+
                            ! Link Run Statistics !
                            +----------------------+

Performance Indicators                   Page Faults   CPU Time        Elapsed Time
----------------------                   -----------   --------        ------------
    Command processing:                           15   00:00:00.11     00:00:00.15
    Pass 1:                                       31   00:00:01.13     00:00:01.51
    Allocation/Relocation:                         3   00:00:00.10     00:00:00.20
    Pass 2:                                       10   00:00:00.52     00:00:00.91
    Map data after object module synopsis:         4   00:00:00.51     00:00:00.83
Total run values:                                 63   00:00:02.37     00:00:03.60

Using a working set limited to 2097151 pages and 235 pages of data storage
    (including image)

Total number object records read (both passes):             243
    of which 108 were in libraries and 1 were DEBUG data records containing
    77 bytes

Number of modules extracted to resolve undefined symbols:   18

❶ lk /lib/crt0.o test.obj -K /usr/lib/fortrtl.a -lc
```

Key to Example 2–9:

❶ The command line used to invoke the lk linker.